Polyvariant program specialisation with property-based abstraction

John P Gallagher

Roskilde University, Denmark

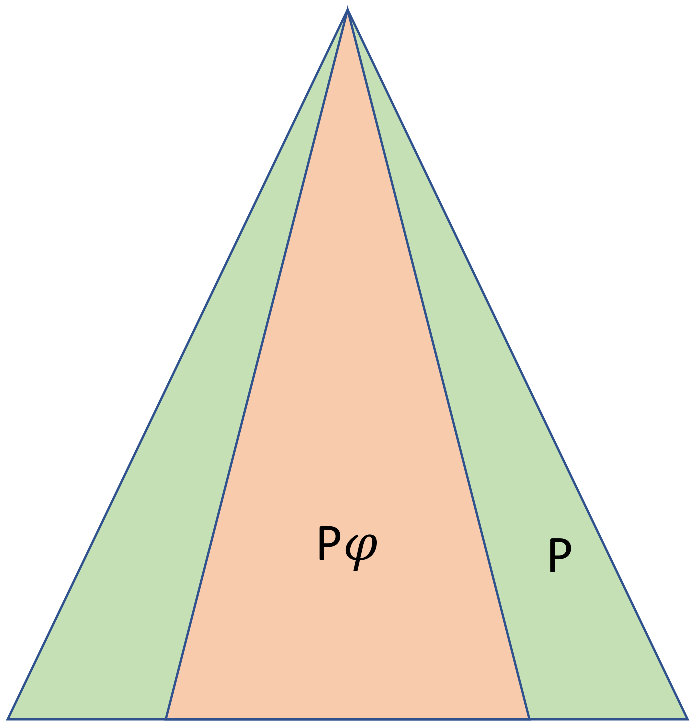
IMDEA Software Institute, Madrid, Spain

VPT 2019, Genoa

Program specialization

- Given a program P
- Let φ be some set of input states for P
- Transform P to P_φ that "behaves the same" as P when starting from a φ-state

- For other initial states, P_φ can be undefined.



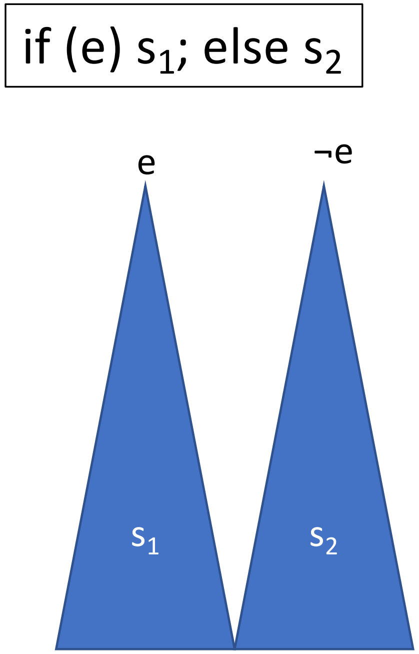
Specialization as optimization

- The aim of the transformation is to gain <u>efficiency</u>.
  - Exploit the knowledge of φ throughout the computation
- Trivial specialization not acceptable

```
if (initial state in φ) {
        call(P);
}
else {
        undefined;
}
```

Internal specialization

- In the paper, focus on internal specialization
- Related to “driving”, equivalent to partial deduction [Glück 1994]

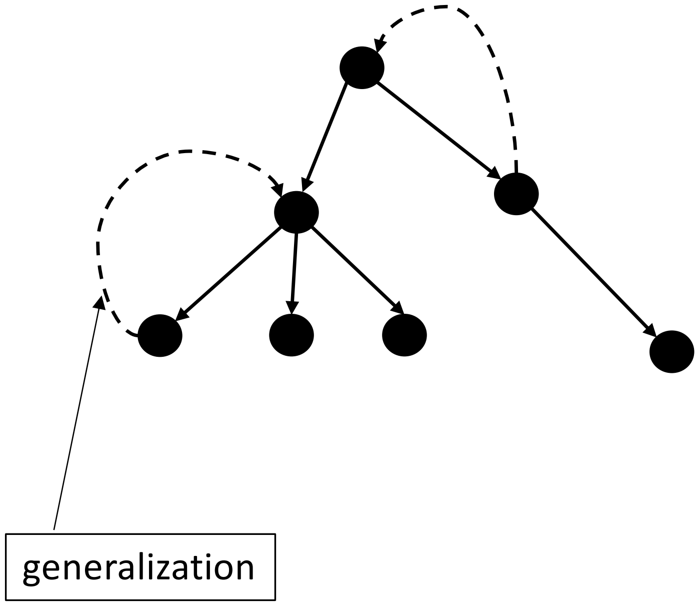


Logic program specialization: 3 approaches

- Global computation tree (e.g. Mixtus)
- Compute set of predicate calls that is “closed” with respect to an unfolding rule (Lloyd-Shepherdson)
- Local unfold-fold-newdef transformations (Pettorossi-Proietti et al.)

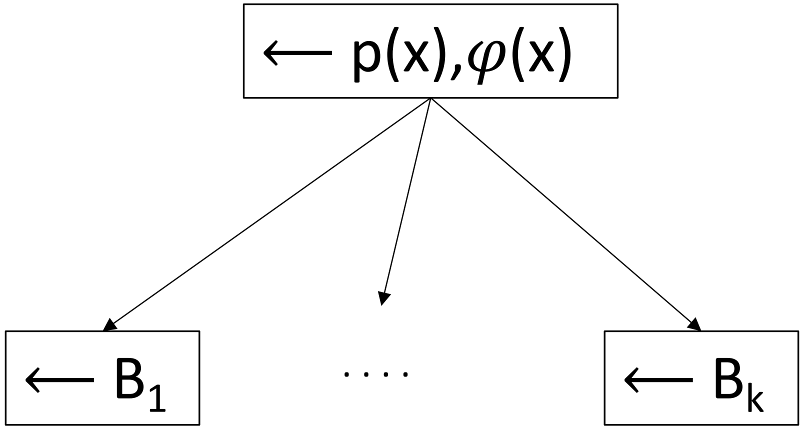
Common aspects in all approaches

- A generalization operation
  - Needed to ensure termination of the specialization algorithm
- A closure property
  - Global tree approaches – branches loop back to ancestors
  - Fold-unfold-newdef - folding wrt to a previously unfolded new definition.



Lloyd-Shepherdson approach

- The algorithm computes a set of predicate calls
- Represent a call to predicate p(x) with constraint φ(x) as
  $p(x) \longleftarrow \varphi(x)$
- Let U be an unfolding rule that builds a partial derivation tree for a call.



The partial evaluation of $p(x) \longleftarrow \varphi(x)$ under U is the set of clauses

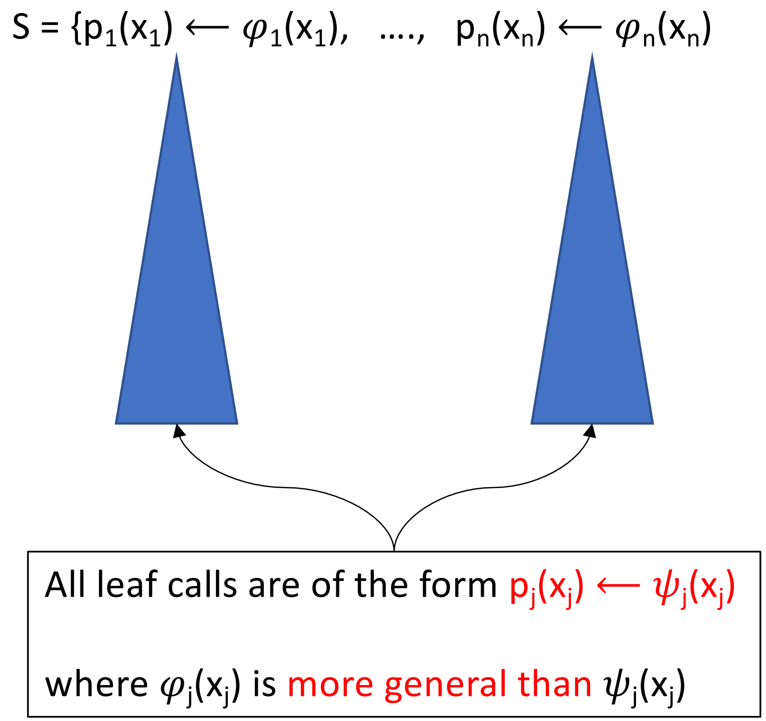
$p(x) \longleftarrow B_1$,

....,

$p(x) \longleftarrow B_k$

Closed set of calls under unfolding rule U

- Given a finite set of calls S and an unfolding rule U
- S is closed under U if the partial evaluation trees for elements of S contain leaves that are subsumed by elements of S



Algorithm to generate a closed set of calls

$S \leftarrow S_0$

repeat

$\quad S' = S$

$\quad S \leftarrow S \cup \alpha_\rho(\mathsf{collect}(\mathsf{pe}(S)))$

until $S' = S$

See Section 3 of paper.
Algorithm structure based on [Gallagher 1993] which followed the Lloyd-Shepherdson approach.

- Start with the initial calls S_0
- Repeat
  - pe: Partially evaluate the set of calls
  - collect: Collect the leaves of the partial trees.
  - α_ρ: Generalise them
- Until the set of calls is closed

The generalization operation

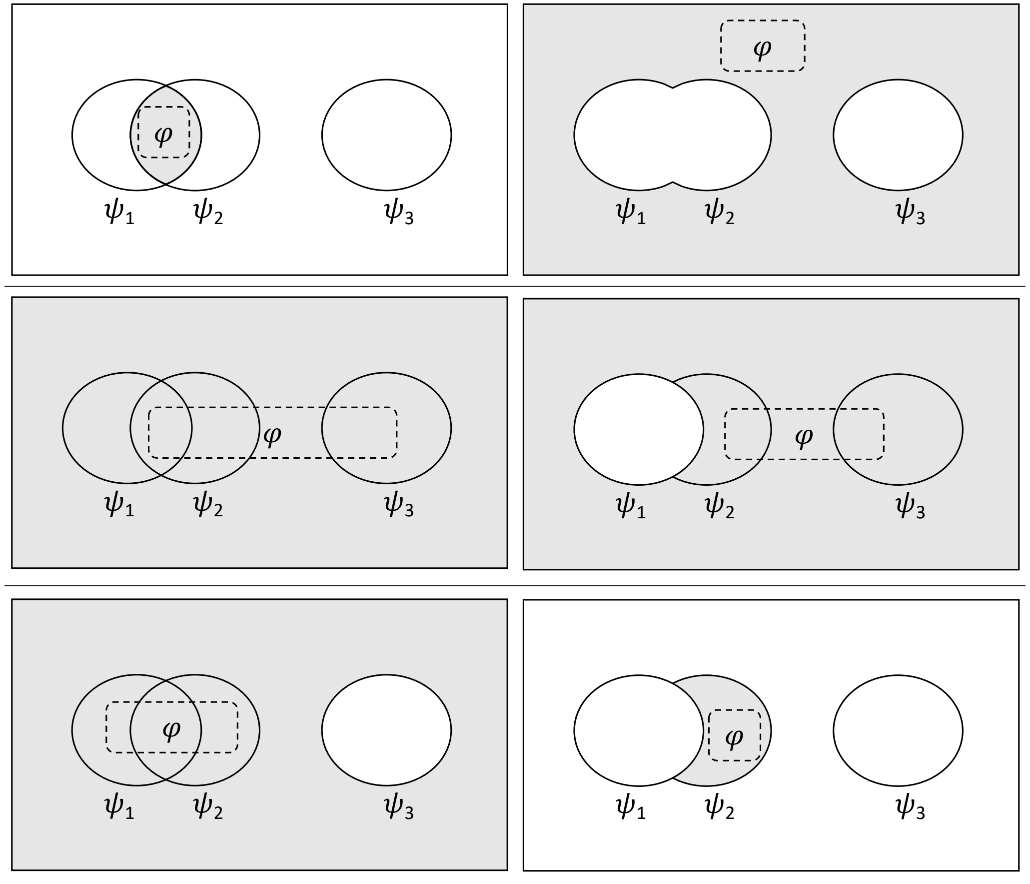
- The generalization operation is crucial
  - Over-generalization – loses specialization
  - Under-generalization – risk code explosion
- In the paper we explore a generalization operation using property-based abstraction

Property-based abstraction

- The idea originated in software model checking [Ball et al. 2001]
- Let p(x) be a predicate and let $\Psi = \{\varphi_1(x), \ldots, \varphi_k(x)\}$ be a finite set of properties

- The property-based abstraction of $\psi(x)$ wrt Ψ is the conjunction of the set of elements of Ψ and their negations that are entailed by $\psi(x)$

Property-based abstraction in pictures



$\Psi = \{\psi_1, \psi_2, \psi_3\}$

φ the property to be abstracted (dotted area)

Shaded area is the result of abstracting φ using Ψ.

It is always a generalization (i.e. a larger area)

Note that only a finite number of different generalizations are possible.

Control-flow refinement

- Why is property-based abstraction a good idea?

- Because the properties chosen for Ψ can be those that determine control-flow in the program

- Consider Example 2 from paper

```
while (x>0) {
  if (y<m)  y++; else x--;
}
```

then branch of **if** statement does not affect **while** condition

else branch of **if** statement does not affect **if** condition

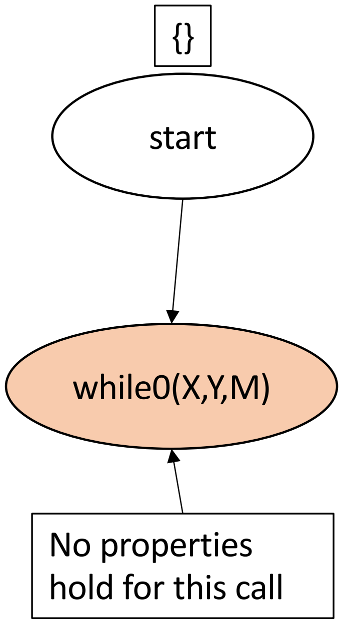
Horn clause representation of program

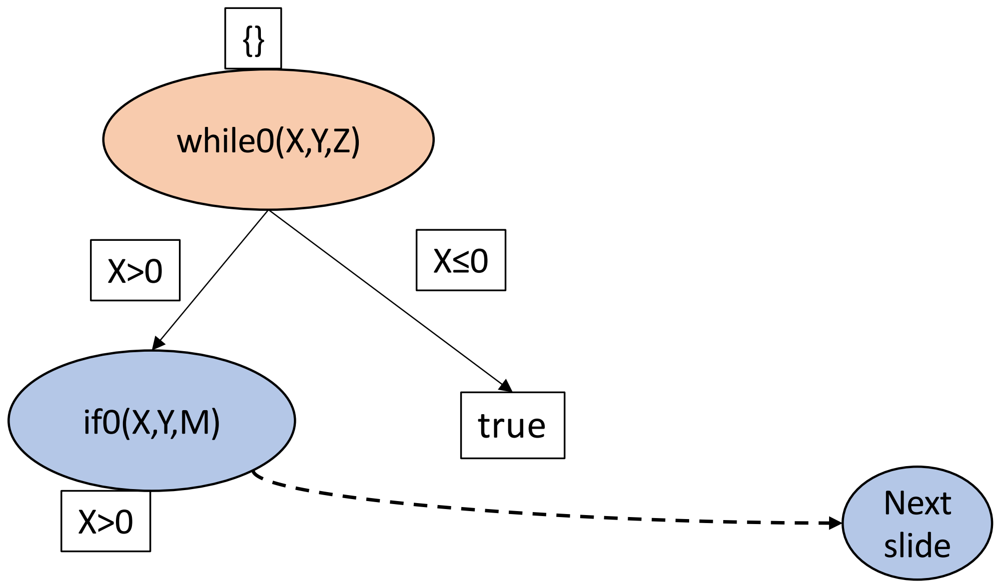
```
start ⟵
        while0(X,Y,M).
while0(X,Y,M)  ⟵
        X>0,
        if0(X,Y,M).
while0(X,Y,M)  ⟵
        X=<0.
if0(X,Y,M)  ⟵
        Y<M, Y1=Y+1,
        while0(X,Y1,M).
if0(X,Y,M)  ⟵
        Y>=M, X1=X-1,
        while0(X1,Y,M).
```

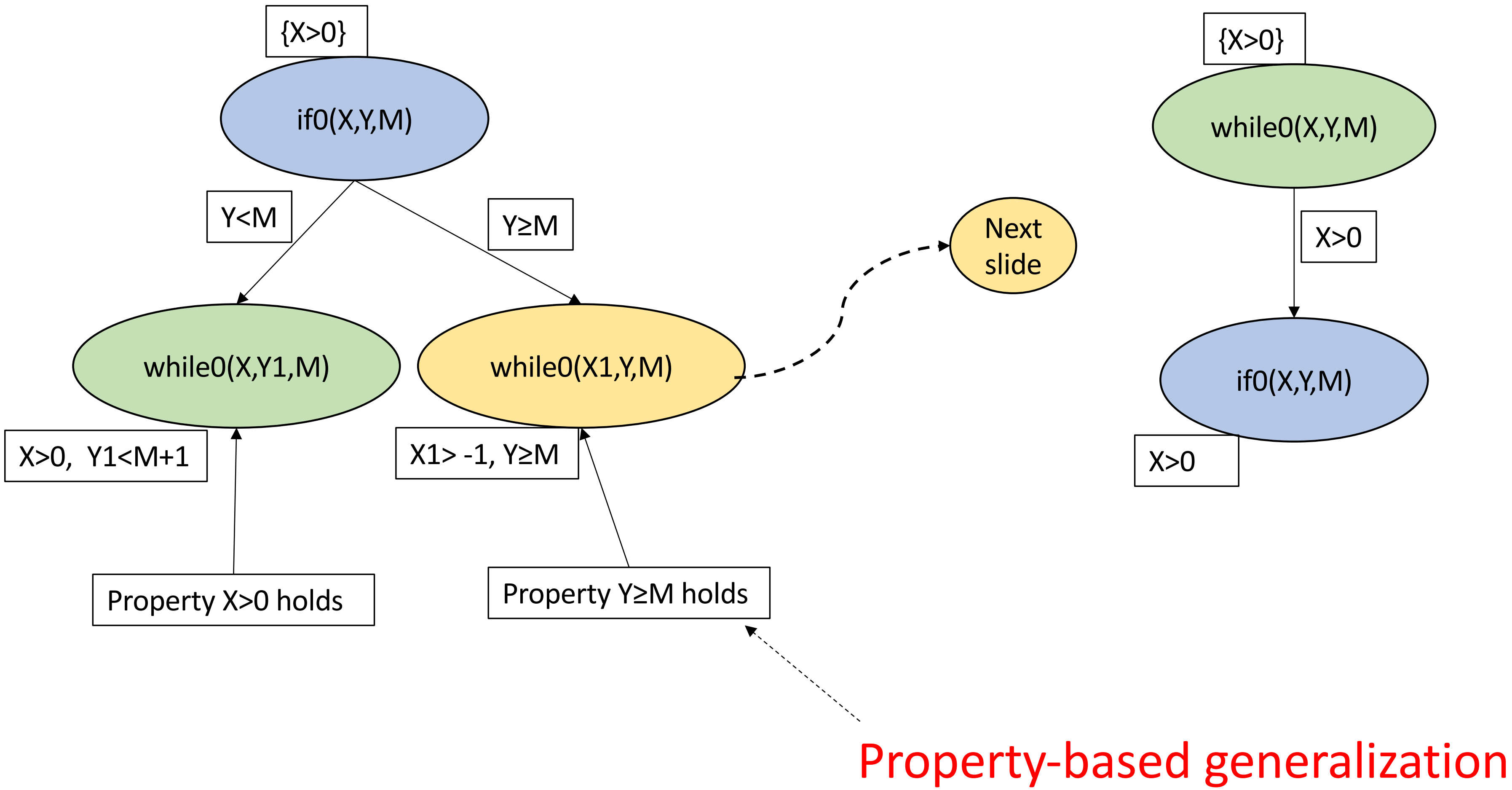
Specialize wrt call to start and the following set of properties

```
1: while0(A,B,C) ⟵  A>0
2: while0(A,B,C) ⟵  A≤0
3: while0(A,B,C) ⟵  B<C
4: while0(A,B,C) ⟵  B≥C
5: if0(A,B,C) ⟵  B<C
6: if0(A,B,C) ⟵  B≥C
```

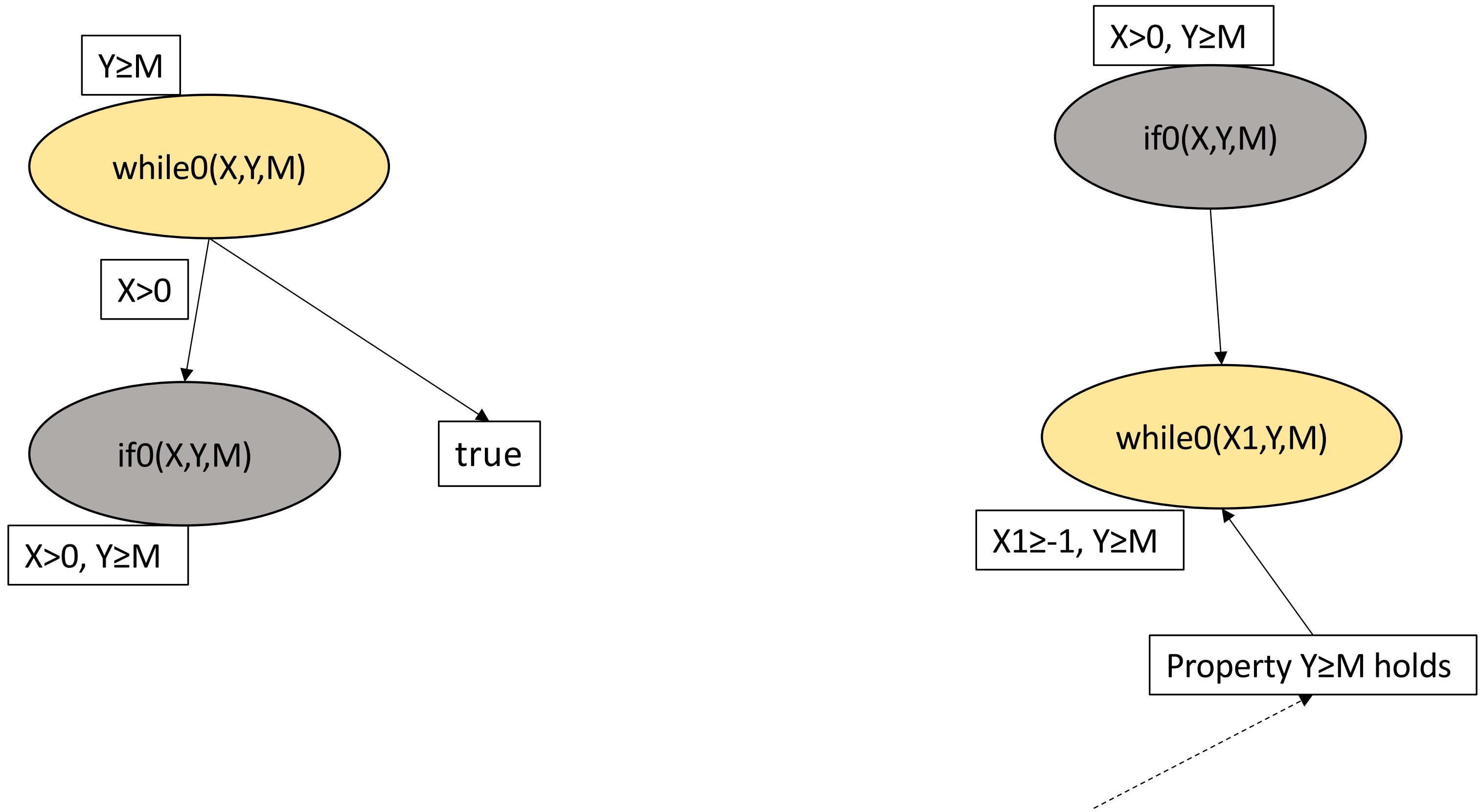
The unfolding rule stops when a branch is reached


{}
start
while0(X,Y,M)
No properties hold for this call


{}
while0(X,Y,Z)
X>0
X≤0
if0(X,Y,M)
true
X>0
Next slide



{X>0}
if0(X,Y,M)
Y<M
Y≥M
while0(X,Y1,M)
while0(X1,Y,M)
Next slide
X>0, Y1<M+1
X1> -1, Y≥M
Property X>0 holds
Property Y≥M holds
Property-based generalization
{X>0}
while0(X,Y,M)
X>0
if0(X,Y,M)
X>0



Property-based generalization

Closed set achieved

- The set of calls is now closed

if0(A,B,C)⟵ A>0, B≥C

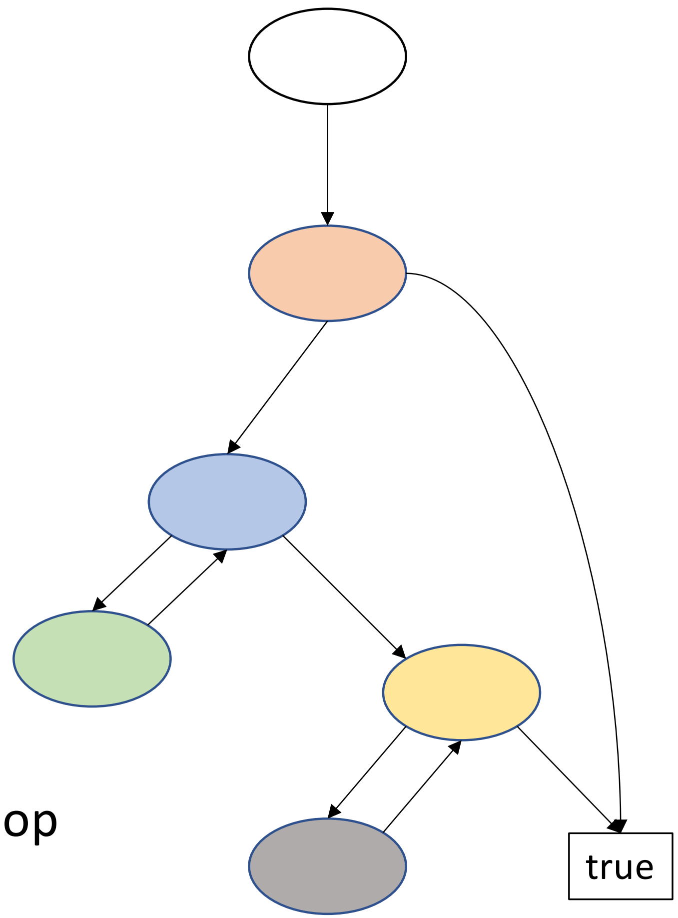
while0(A,B,C) ⟵ B≥C

while0(A,B,C) ⟵ A>0

if0(A,B,C) ⟵ A>0

while0(A,B,C) ⟵ true

start ⟵ true



2-phase loop

Reconstructed imperative code

```
start ⟵  while5(A,B,C).
while5(A,B,C) ⟵   A>0,if4(A,B,C).
while5(A,B,C) ⟵   -A≥0.
if4(A,B,C)   ⟵ A>0,-B+C>0,B+ -D= -1,while3(A,D,C).
if4(A,B,C) ⟵   A>0,B+ -C≥0,A+ -D=1,while2(D,B,C).
while3(A,B,C) ⟵   A>0,if4(A,B,C).
while2(A,B,C) ⟵   B+ -C≥0,A>0,if1(A,B,C).
while2(A,B,C)   ⟵ B+ -C≥0,-A>=0.
if1(A,B,C) ⟵   A>0,B+ -C≥0,A+ -D=1,while2(D,B,C).
```

```
if (x>0) {
  while (y<m) { /* x>0 */
    y++;}
  x--;
  while (x>0) { /* y>=m */
    x--;}
  }
```

2-phase loop

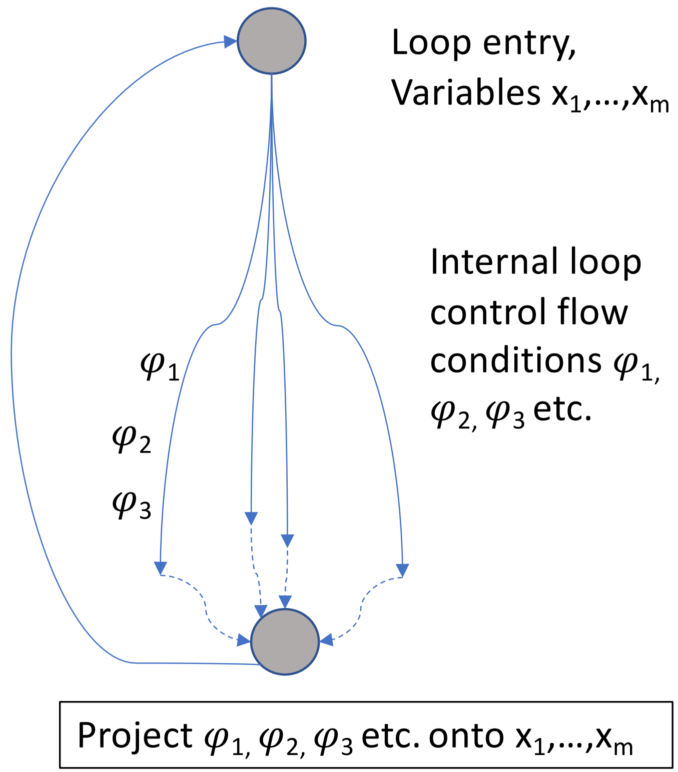
Polyvariant specialization

- Polyvariant specialization means that more than one version of a call is generated.
- Different constraints on calls can result in different control flow
- Experiments show that polyvariant specialization using property-based abstraction improves termination analysis
- E.g. the 2-phase loop is easily proved to be terminating, but the original program is not

What is a good set of properties?

- The key abstractions apply to loop entry points.

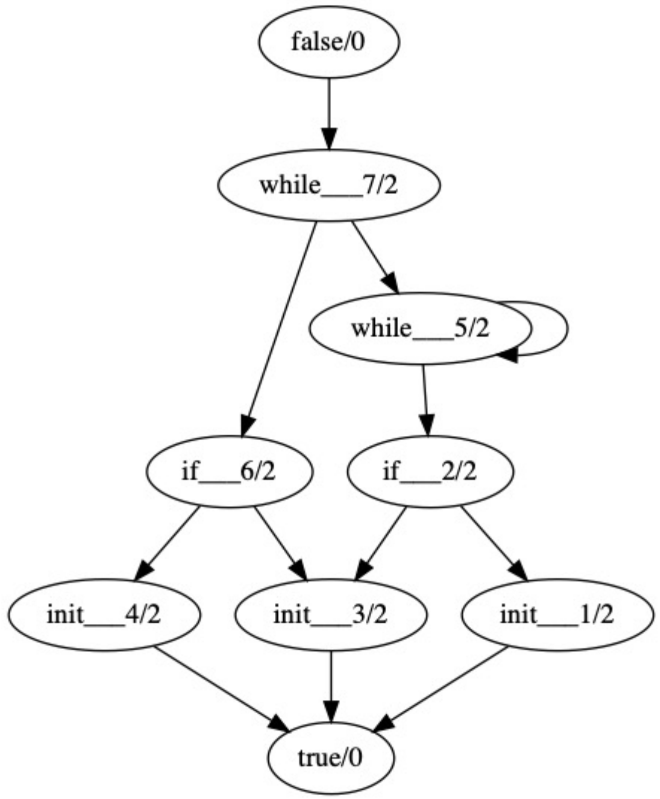
- Consider a loop. Collect all the choices made within the loop, projected onto the variables at the loop entry point
- That is, the properties collect all the relevant information determining which path through the loop will be taken



Polyvariance and disjunctive invariants

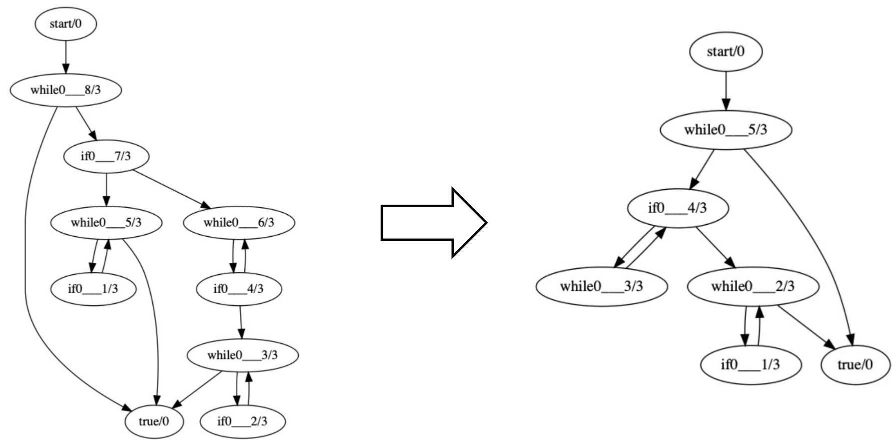
- A good set of loop properties helps to find disjunctive invariants

- Many verification problems require discovery of disjunctive invariants
  - Difficult to achieve automatically using standard abstractions such as convex polyhedra



Control flow minimization

- We may over-specialize a program by choosing properties that are too fine-grained.
- The different versions that are produced may simply be “clones” of each other
- Automata minimization can reduce to the minimum number of versions
- *Tree automata* minimization for non-linear Horn clauses



Is there a set of properties that would generate the minimized version directly?

Conclusions

- Property-based abstraction has many practical advantages as a generalization mechanism in program specialization
  - Easy to implement using a SAT/SMT solver
  - Guarantees termination of specialization
  - Relevant properties can be generated beforehand, capturing control-flow
  - (but more study is needed on this)
- We can reproduce special-purpose techniques from the literature, regarding control-flow refinement and loop splitting
- Like all finite abstraction technique (with no widening) it can lose precision and potential specializations.